

---

# **The Effects Tutorial**

*Release 0.9.19.1*

**The Idris Community**

November 12, 2015



<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Hello world . . . . .	3
1.2	Outline . . . . .	4
<b>2</b>	<b>State</b>	<b>5</b>
2.1	First attempt . . . . .	5
2.2	Introducing <code>Effects</code> . . . . .	6
2.3	Effects and Resources . . . . .	8
2.4	Labelled Effects . . . . .	8
2.5	!-notation . . . . .	9
2.6	The Type <code>Eff</code> . . . . .	10
<b>3</b>	<b>Simple Effects</b>	<b>13</b>
3.1	Console I/O . . . . .	13
3.2	Exceptions . . . . .	14
3.3	Random Numbers . . . . .	16
3.4	Non-determinism . . . . .	17
3.5	<code>vadd</code> revisited . . . . .	17
3.6	Example: An Expression Calculator . . . . .	18
<b>4</b>	<b>Dependent Effects</b>	<b>21</b>
4.1	Dependent States . . . . .	21
4.2	Result-dependent Effects . . . . .	21
4.3	File Management . . . . .	22
4.4	Pattern-matching <code>bind</code> . . . . .	24
<b>5</b>	<b>Creating New Effects</b>	<b>27</b>
5.1	State . . . . .	27
5.2	Console I/O . . . . .	29
5.3	Exceptions . . . . .	30
5.4	Non-determinism . . . . .	30
5.5	File Management . . . . .	30
<b>6</b>	<b>Example: A “Mystery Word” Guessing Game</b>	<b>33</b>
6.1	Step 1: Game State . . . . .	33
6.2	Step 2: Game Rules . . . . .	33
6.3	Step 3: Implement Rules . . . . .	34
6.4	Step 4: Implement Interface . . . . .	35
6.5	Discussion . . . . .	36
<b>7</b>	<b>Further Reading</b>	<b>39</b>
<b>8</b>	<b>Effects Summary</b>	<b>41</b>

8.1	EXCEPTION	41
8.2	FILE_IO	41
8.3	RND	42
8.4	SELECT	42
8.5	STATE	42
8.6	STDIO	42
8.7	SYSTEM	43

A tutorial on the *Effects* package in *Idris*.

---

**Note:** The documentation for Idris has been published under the Creative Commons CC0 License. As such to the extent possible under law, *The Idris Community* has waived all copyright and related or neighbouring rights to Documentation for Idris.

More information concerning the CC0 can be found online at: <http://creativecommons.org/publicdomain/zero/1.0/>

---



---

## Introduction

---

Pure functional languages with dependent types such as `Idris` support reasoning about programs directly in the type system, promising that we can *know* a program will run correctly (i.e. according to the specification in its type) simply because it compiles. Realistically, though, things are not so simple: programs have to interact with the outside world, with user input, input from a network, mutable state, and so on. In this tutorial I will introduce the library, which is included with the distribution and supports programming and reasoning with side-effecting programs, supporting mutable state, interaction with the outside world, exceptions, and verified resource management.

This tutorial assumes familiarity with pure programming in `Idris`, as described in Sections 1–6 of the main tutorial <sup>1</sup>. The examples presented are tested with `Idris` and can be found in the `examples` directory of the `Idris` repository.

Consider, for example, the following introductory function which illustrates the kind of properties which can be expressed in the type system:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

This function adds corresponding elements in a pair of vectors. The type guarantees that the vectors will contain only elements of type `Int`, and that the input lengths and the output length all correspond. A natural question to ask here, which is typically neglected by introductory tutorials, is “How do I turn this into a program?” That is, given some lists entered by a user, how do we get into a position to be able to apply the `vadd` function? Before doing so, we will have to:

- Read user input, either from the keyboard, a file, or some other input device.
- Check that the user inputs are valid, i.e. contain only `Int` and are the same length, and report an error if not.
- Write output

The complete program will include side-effects for I/O and error handling, before we can get to the pure core functionality. In this tutorial, we will see how `Idris` supports side-effects. Furthermore, we will see how we can use the dependent type system to *reason* about stateful and side-effecting programs. We will return to this specific example later.

### 1.1 Hello world

To give an idea of how programs with effects look in `Idris`, here is the ubiquitous “Hello world” program, written using the `Effects` library:

```
module Main
import Effects
```

<sup>1</sup> You do not, however, need to know what a monad is!

```
import Effect.StdIO

hello : Eff () [STDIO]
hello = putStrLn "Hello world!"

main : IO ()
main = run hello
```

As usual, the entry point is `main`. All `main` has to do is invoke the `hello` function which supports the `STDIO` effect for console I/O, and returns the unit value. All programs using the `Effects` library must import `Effects`. The details of the `Eff` type will be presented in the remainder of this tutorial.

To compile and run this program, `Idris` needs to be told to include the `Effects` package, using the `-p effects` flag (this flag is required for all examples in this tutorial):

```
idris hello.idr -o hello -p effects
./hello Hello world!
```

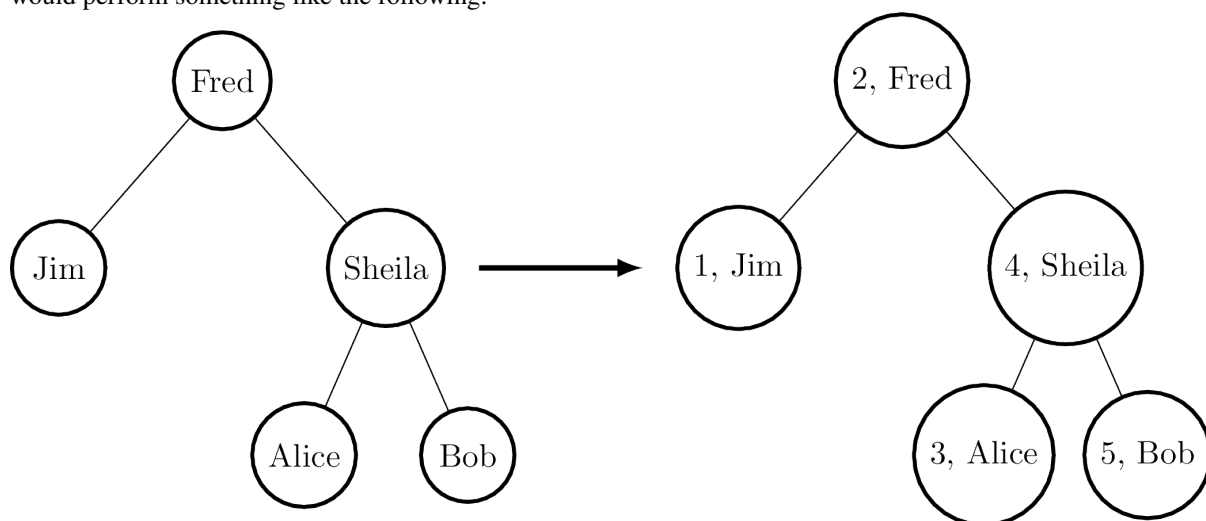
## 1.2 Outline

The tutorial is structured as follows: first, in Section *State*, we will discuss state management, describing why it is important and introducing the `effects` library to show how it can be used to manage state. This section also gives an overview of the syntax of effectful programs. Section *Simple Effects* then introduces a number of other effects a program may have: I/O; Exceptions; Random Numbers; and Non-determinism, giving examples for each, and an extended example combining several effects in one complete program. Section *Dependent Effects* introduces *dependent* effects, showing how states and resources can be managed in types. Section *Creating New Effects* shows how new effects can be implemented. Section *Example: A “Mystery Word” Guessing Game* gives an extended example showing how to implement a “mystery word” guessing game, using effects to describe the rules of the game and ensure they are implemented accurately. References to further reading are given in Section *Further Reading*.



## State

Many programs, even pure programs, can benefit from locally mutable state. For example, consider a program which tags binary tree nodes with a counter, by an inorder traversal (i.e. counting depth first, left to right). This would perform something like the following:



We can describe binary trees with the following data type `BTree` and `testTree` to represent the example input above:

```
data BTree a = Leaf
             | Node (BTree a) a (BTree a)

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
               "Fred"
               (Node (Node Leaf "Alice" Leaf)
                     "Sheila"
                     (Node Leaf "Bob" Leaf))
```

Then our function to implement tagging, beginning to tag with a specific value `i`, has the following type:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
```

## 2.1 First attempt

Naïvely, we can implement `treeTag` by implementing a helper function which propagates a counter, returning the result of the count for each subtree:

```
treeTagAux : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTagAux i Leaf = (i, Leaf)
```

```
treeTagAux i (Node l x r)
  = let (i', l') = treeTagAux i l in
      let x' = (i', x) in
          let (i'', r') = treeTagAux (i' + 1) r in
              (i'', Node l' x' r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = snd (treeTagAux i x)
```

This gives the expected result when run at the REPL prompt:

```
*TreeTag> treeTag 1 testTree
Node (Node Leaf (1, "Jim") Leaf)
      (2, "Fred")
      (Node (Node Leaf (3, "Alice") Leaf)
            (4, "Sheila")
            (Node Leaf (5, "Bob") Leaf)) : BTree (Int, String)
```

This works as required, but there are several problems when we try to scale this to larger programs. It is error prone, because we need to ensure that state is propagated correctly to the recursive calls (i.e. passing the appropriate `i` or `i'`). It is hard to read, because the functional details are obscured by the state propagation. Perhaps most importantly, there is a common programming pattern here which should be abstracted but instead has been implemented by hand. There is local mutable state (the counter) which we have had to make explicit.

## 2.2 Introducing Effects

Idris provides a library, `Effects`<sup>1</sup>, which captures this pattern and many others involving effectful computation<sup>2</sup>. An effectful program `f` has a type of the following form:

```
f : (x1 : a1) -> (x2 : a2) -> ... -> Eff t effs
```

That is, the return type gives the effects that `f` supports (`effs`, of type `List EFFECT`) and the type the computation returns `t`. So, our `treeTagAux` helper could be written with the following type:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
```

That is, `treeTagAux` has access to an integer state, because the list of available effects includes `STATE Int`. `STATE` is declared as follows in the module `Effect.State` (that is, we must import `Effect.State` to be able to use it):

```
STATE : Type -> EFFECT
```

It is an effect parameterised by a type (by convention, we write effects in all capitals). The `treeTagAux` function is an effectful program which builds a new tree tagged with `Ints`, and is implemented as follows:

```
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r)
  = do l' <- treeTagAux l
      i <- get
      put (i + 1)
      r' <- treeTagAux r
      pure (Node l' (i, x) r')
```

There are several remarks to be made about this implementation. Essentially, it hides the state, which can be accessed using `get` and updated using `put`, but it introduces several new features. Specifically, it uses `do`-notation, binding variables with `<-`, and a `pure` function. There is much to be said about these features, but for our purposes, it suffices to know the following:

<sup>1</sup> Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 <http://dl.acm.org/citation.cfm?doid=2544174.2500581>

<sup>2</sup> The earlier paper<sup>3</sup> describes the essential implementation details, although the library presented there is an earlier version which is less powerful than that presented in this tutorial.

- `do` blocks allow effectful operations to be sequenced.
- `x <- e` binds the result of an effectful operation `e` to a variable `x`. For example, in the above code, `treeTagAux l` is an effectful operation returning `BTree (Int, a)`, so `l'` has type `BTree (Int, a)`.
- `pure e` turns a pure value `e` into the result of an effectful operation.

The `get` and `put` functions read and write a state `t`, assuming that the `STATE t` effect is available. They have the following types, polymorphic in the state `t` they manage:

```
get :      Eff t [STATE t]
put : t -> Eff () [STATE t]
```

A program in `Eff` can call any other function in `Eff` provided that the calling function supports at least the effects required by the called function. In this case, it is valid for `treeTagAux` to call both `get` and `put` because all three functions support the `STATE Int` effect.

Programs in `Eff` are run in some underlying *computation context*, using the `run` or `runPure` function. Using `runPure`, which runs an effectful program in the identity context, we can write the `treeTag` function as follows, using `put` to initialise the state:

```
treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i
                          treeTagAux x)
```

We could also run the program in an impure context such as `IO`, without changing the definition of `treeTagAux`, by using `run` instead of `runPure`:

```
treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
...
treeTag : (i : Int) -> BTree a -> IO (BTree (Int, a))
treeTag i x = run (do put i
                      treeTagAux x)
```

Note that the definition of `treeTagAux` is exactly as before. For reference, this complete program (including a `main` to run it) is shown in Listing [introprog].

```
module Main

import Effects
import Effect.State

data BTree a = Leaf
             | Node (BTree a) a (BTree a)

instance Show a => Show (BTree a) where
  show Leaf = "[]"
  show (Node l x r) = "[" ++ show l ++ " "
                    ++ show x ++ " "
                    ++ show r ++ "]"

testTree : BTree String
testTree = Node (Node Leaf "Jim" Leaf)
              "Fred"
              (Node (Node Leaf "Alice" Leaf)
                  "Sheila"
                  (Node Leaf "Bob" Leaf))

treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE Int]
treeTagAux Leaf = pure Leaf
treeTagAux (Node l x r) = do l' <- treeTagAux l
                             i <- get
                             put (i + 1)
```

```

        r' <- treeTagAux r
        pure (Node l' (i, x) r')

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPure (do put i; treeTagAux x)

main : IO ()
main = print (treeTag 1 testTree)

```

## 2.3 Effects and Resources

Each effect is associated with a *resource*, which is initialised before an effectful program can be run. For example, in the case of `STATE Int` the corresponding resource is the integer state itself. The types of `runPure` and `run` show this (slightly simplified here for illustrative purposes):

```

runPure : {env : Env id xs} -> Eff a xs -> a
run : Applicative m => {env : Env m xs} -> Eff a xs -> m a

```

The `env` argument is implicit, and initialised automatically where possible using default values given by instances of the following type class:

```

class Default a where
  default : a

```

Instances of `Default` are defined for all primitive types, and many library types such as `List`, `Vect`, `Maybe`, pairs, etc. However, where no default value exists for a resource type (for example, you may want a `STATE` type for which there is no `Default` instance) the resource environment can be given explicitly using one of the following functions:

```

runPureInit : Env id xs -> Eff a xs -> a
runInit : Applicative m => Env m xs -> Eff a xs -> m a

```

To be well-typed, the environment must contain resources corresponding exactly to the effects in `xs`. For example, we could also have implemented `treeTag` by initialising the state as follows:

```

treeTag : (i : Int) -> BTree a -> BTree (Int, a)
treeTag i x = runPureInit [i] (treeTagAux x)

```

## 2.4 Labelled Effects

What if we have more than one state, especially more than one state of the same type? How would `get` and `put` know which state they should be referring to? For example, how could we extend the tree tagging example such that it additionally counts the number of leaves in the tree? One possibility would be to change the state so that it captured both of these values, e.g.:

```

treeTagAux : BTree a -> Eff (BTree (Int, a)) [STATE (Int, Int)]

```

Doing this, however, ties the two states together throughout (as well as not indicating which integer is which). It would be nice to be able to call effectful programs which guaranteed only to access one of the states, for example. In a larger application, this becomes particularly important.

The library therefore allows effects in general to be *labelled* so that they can be referred to explicitly by a particular name. This allows multiple effects of the same type to be included. We can count leaves and update the tag separately, by labelling them as follows:

```

treeTagAux : BTree a -> Eff (BTree (Int, a))
              ['Tag :: STATE Int,
               'Leaves :: STATE Int]

```

The `:::` operator allows an arbitrary label to be given to an effect. This label can be any type—it is simply used to identify an effect uniquely. Here, we have used a symbol type. In general `' name` introduces a new symbol, the only purpose of which is to disambiguate values<sup>3</sup>.

When an effect is labelled, its operations are also labelled using the `:-` operator. In this way, we can say explicitly which state we mean when using `get` and `put`. The tree tagging program which also counts leaves can be written as follows:

```
treeTagAux Leaf = do
  'Leaves :- update (+1)
  pure Leaf
treeTagAux (Node l x r) = do
  l' <- treeTagAux l
  i <- 'Tag :- get
  'Tag :- put (i + 1)
  r' <- treeTagAux r
  pure (Node l' (i, x) r')
```

The update function here is a combination of `get` and `put`, applying a function to the current state.

```
update : (x -> x) -> Eff () [STATE x]
```

Finally, our top level `treeTag` function now returns a pair of the number of leaves, and the new tree. Resources for labelled effects are initialised using the `:=` operator (reminiscent of assignment in an imperative language):

```
treeTag : (i : Int) -> BTree a -> (Int, BTree (Int, a))
treeTag i x = runPureInit ['Tag := i, 'Leaves := 0]
              (do x' <- treeTagAux x
                 leaves <- 'Leaves :- get
                 pure (leaves, x'))
```

To summarise, we have:

- `:::` to convert an effect to a labelled effect.
- `:-` to convert an effectful operation to a labelled effectful operation.
- `:=` to initialise a resource for a labelled effect.

Or, more formally with their types (slightly simplified to account only for the situation where available effects are not updated):

```
(:::) : lbl -> EFFECT -> EFFECT
(:-)  : (l : lbl) -> Eff a [x] -> Eff a [l ::: x]
(:=)  : (l : lbl) -> res -> LRes l res
```

Here, `LRes` is simply the resource type associated with a labelled effect. Note that labels are polymorphic in the label type `lbl`. Hence, a label can be anything—a string, an integer, a type, etc.

## 2.5 !-notation

In many cases, using `do`-notation can make programs unnecessarily verbose, particularly in cases where the value bound is used once, immediately. The following program returns the length of the `String` stored in the state, for example:

```
stateLength : Eff Nat [STATE String]
stateLength = do x <- get
              pure (length x)
```

This seems unnecessarily verbose, and it would be nice to program in a more direct style in these cases. provides `!`-notation to help with this. The above program can be written instead as:

<sup>3</sup> In practice, `' name` simply introduces a new empty type

```
stateLength : Eff Nat [STATE String]
stateLength = pure (length !get)
```

The notation `!expr` means that the expression `expr` should be evaluated and then implicitly bound. Conceptually, we can think of `!` as being a prefix function with the following type:

```
(!) : Eff a xs -> a
```

Note, however, that it is not really a function, merely syntax! In practice, a subexpression `!expr` will lift `expr` as high as possible within its current scope, bind it to a fresh name `x`, and replace `!expr` with `x`. Expressions are lifted depth first, left to right. In practice, `!`-notation allows us to program in a more direct style, while still giving a notational clue as to which expressions are effectful.

For example, the expression:

```
let y = 42 in f !(g !(print y) !x)
```

is lifted to:

```
let y = 42 in do y' <- print y
                x' <- x
                g' <- g y' x'
                f g'
```

## 2.6 The Type `Eff`

Underneath, `Eff` is an overloaded function which translates to an underlying type `EffM`:

```
EffM : (m : Type -> Type) -> (t : Type)
      -> (List EFFECT)
      -> (t -> List EFFECT) -> Type
```

This is more general than the types we have been writing so far. It is parameterised over an underlying computation context `m`, a result type `t` as we have already seen, as well as a `List EFFECT` and a function type `t -> List EFFECT`.

These additional parameters are the list of *input* effects, and a list of *output* effects, computed from the result of an effectful operation. That is: running an effectful program can change the set of effects available! This is a particularly powerful idea, and we will see its consequences in more detail later. Some examples of operations which can change the set of available effects are:

- Updating a state containing a dependent type (for example adding an element to a vector).
- Opening a file for reading is an effect, but whether the file really *is* open afterwards depends on whether the file was successfully opened.
- Closing a file means that reading from the file should no longer be possible.

While powerful, this can make uses of the `EffM` type hard to read. Therefore the library provides an overloaded function `Eff`. There are the following three versions:

```
SimpleEff.Eff : (t : Type) -> (input_effs : List EFFECT) -> Type
TransEff.Eff  : (t : Type) -> (input_effs : List EFFECT) ->
                    (output_effs : List EFFECT) -> Type
DepEff.Eff    : (t : Type) -> (input_effs : List EFFECT) ->
                    (output_effs_fn : x -> List EFFECT) -> Type
```

So far, we have used only the first version, `SimpleEff.Eff`, which is defined as follows:

```
Eff : (x : Type) -> (es : List EFFECT) -> Type
Eff x es = {m : Type -> Type} -> EffM m x es (\v => es)
```

i.e. the set of effects remains the same on output. This suffices for the `STATE` example we have seen so far, and for many useful side-effecting programs. We could also have written `treeTagAux` with the expanded type:

```
treeTagAux : BTree a ->
             EffM m (BTree (Int, a)) [STATE Int] (\x => [STATE Int])
```

Later, we will see programs which update effects:

```
Eff a xs xs'
```

which is expanded to

```
EffM m a xs (\_ => xs')
```

i.e. the set of effects is updated to `xs'` (think of a transition in a state machine). There is, for example, a version of `put` which updates the type of the state:

```
putM : y -> Eff () [STATE x] [STATE y]
```

Also, we have:

```
Eff t xs (\res => xs')
```

which is expanded to

```
EffM m t xs (\res => xs')
```

i.e. the set of effects is updated according to the result of the operation `res`, of type `t`.

Parameterising `EffM` over an underlying computation context allows us to write effectful programs which are specific to one context, and in some cases to write programs which *extend* the list of effects available using the new function, though this is beyond the scope of this tutorial.





---

## Simple Effects

---

So far we have seen how to write programs with locally mutable state using the `STATE` effect. To recap, we have the definitions below in a module `Effect.State`

```

module Effect.State

STATE : Type -> EFFECT

get    :                      Eff x  [STATE x]
put    : x ->                 Eff () [STATE x]
putM   : y ->                 Eff () [STATE x] [STATE y]
update : (x -> x) -> Eff () [STATE x]

instance Handler State m

```

The last line, `instance Handler State m`, means that the `STATE` effect is usable in any computation context `m`. That is, a program which uses this effect and returns something of type `a` can be evaluated to something of type `m a` using `run`, for any `m`. The lower case `State` is a data type describing the operations which make up the `STATE` effect itself—we will go into more detail about this in Section *Creating New Effects*.

In this section, we will introduce some other supported effects, allowing console I/O, exceptions, random number generation and non-deterministic programming. For each effect we introduce, we will begin with a summary of the effect, its supported operations, and the contexts in which it may be used, like that above for `STATE`, and go on to present some simple examples. At the end, we will see some examples of programs which combine multiple effects.

All of the effects in the library, including those described in this section, are summarised in Appendix *Effects Summary*.

### 3.1 Console I/O

Console I/O is supported with the `STDIO` effect, which allows reading and writing characters and strings to and from standard input and standard output. Notice that there is a constraint here on the computation context `m`, because it only makes sense to support console I/O operations in a context where we can perform (or at the very least simulate) console I/O:

```

module Effect.StdIO

STDIO : EFFECT

putChar  : Char -> Eff () [STDIO]
putStr   : String -> Eff () [STDIO]
putStrLn : String -> Eff () [STDIO]

getStr   :                      Eff String [STDIO]
getChar  :                      Eff Char  [STDIO]

```

```
instance Handler StdIO IO
instance Handler StdIO (IOExcept a)
```

### 3.1.1 Examples

A program which reads the user's name, then says hello, can be written as follows:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
          x <- getStr
          putStrLn ("Hello " ++ trim x ++ "!")
```

We use `trim` here to remove the trailing newline from the input. The resource associated with `STDIO` is simply the empty tuple, which has a default value `()`, so we can run this as follows:

```
main : IO ()
main = run hello
```

In `hello` we could also use `!`-notation instead of `x <- getStr`, since we only use the string that is read once:

```
hello : Eff () [STDIO]
hello = do putStr "Name? "
          putStrLn ("Hello " ++ trim !getStr ++ "!")
```

More interestingly, we can combine multiple effects in one program. For example, we can loop, counting the number of people we've said hello to:

```
hello : Eff () [STATE Int, STDIO]
hello = do putStr "Name? "
          putStrLn ("Hello " ++ trim !getStr ++ "!")
          update (+1)
          putStrLn ("I've said hello to: " ++ show !get ++ " people")
          hello
```

The list of effects given in `hello` means that the function can call `get` and `put` on an integer state, and any functions which read and write from the console. To run this, `main` does not need to be changed.

### 3.1.2 Aside: Resource Types

To find out the resource type of an effect, if necessary (for example if we want to initialise a resource explicitly with `runInit` rather than using a default value with `run`) we can run the `resourceType` function at the REPL:

```
*ConsoleIO> resourceType STDIO
() : Type
*ConsoleIO> resourceType (STATE Int)
Int : Type
```

## 3.2 Exceptions

The `EXCEPTION` effect is declared in module `Effect.Exception`. This allows programs to exit immediately with an error, or errors to be handled more generally:

```
module Effect.Exception

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]
```

```
instance Handler (Exception a) Maybe
instance Handler (Exception a) List
instance Handler (Exception a) (Either a)
instance Handler (Exception a) (IOExcept a)
instance Show a => Handler (Exception a) IO
```

### 3.2.1 Example

Suppose we have a `String` which is expected to represent an integer in the range 0 to `n`. We can write a function `parseNumber` which returns an `Int` if parsing the string returns a number in the appropriate range, or throws an exception otherwise. Exceptions are parameterised by an error type:

```
data Err = NotANumber | OutOfRange

parseNumber : Int -> String -> Eff Int [EXCEPTION Err]
parseNumber num str
  = if all isDigit (unpack str)
    then let x = cast str in
          if (x >= 0 && x <= num)
            then pure x
            else raise OutOfRange
    else raise NotANumber
```

Programs which support the `EXCEPTION` effect can be run in any context which has some way of throwing errors, for example, we can run `parseNumber` in the `Either Err` context. It returns a value of the form `Right x` if successful:

```
*Exception> the (Either Err Int) $ run (parseNumber 42 "20")
Right 20 : Either Err Int
```

Or `Left e` on failure, carrying the appropriate exception:

```
*Exception> the (Either Err Int) $ run (parseNumber 42 "50")
Left OutOfRange : Either Err Int

*Exception> the (Either Err Int) $ run (parseNumber 42 "twenty")
Left NotANumber : Either Err Int
```

In fact, we can do a little bit better with `parseNumber`, and have it return a *proof* that the integer is in the required range along with the integer itself. One way to do this is define a type of bounded integers, `Bounded`:

```
Bounded : Int -> Type
Bounded x = (n : Int ** So (n >= 0 && n <= x))
```

Recall that `So` is parameterised by a `Bool`, and only `So True` is inhabited. We can use `choose` to construct such a value from the result of a dynamic check:

```
data So : Bool -> Type = Oh : So True

choose : (b : Bool) -> Either (So b) (So (not b))
```

We then write `parseNumber` using `choose` rather than an `if/then/else` construct, passing the proof it returns on success as the boundedness proof:

```
parseNumber : (x : Int) -> String -> Eff (Bounded x) [EXCEPTION Err]
parseNumber x str
  = if all isDigit (unpack str)
    then let num = cast str in
          case choose (num >= 0 && num <= x) of
            Left p => pure (num ** p)
            Right p => raise OutOfRange
    else raise NotANumber
```

## 3.3 Random Numbers

Random number generation is also implemented by the library, in module `Effect.Random`:

```
module Effect.Random

RND : EFFECT

srand  : Integer -> Eff () [RND]
rndInt : Integer -> Integer -> Eff Integer [RND]
rndFin : (k : Nat) -> Eff (Fin (S k)) [RND]

instance Handler Random m
```

Random number generation is considered side-effecting because its implementation generally relies on some external source of randomness. The default implementation here relies on an integer *seed*, which can be set with `srand`. A specific seed will lead to a predictable, repeatable sequence of random numbers. There are two functions which produce a random number:

- **`rndInt`**, which returns a random integer between the given `lower` and upper bounds.
- **`rndFin`**, which returns a random element of a finite set (essentially a number with an upper bound given in its type).

### 3.3.1 Example

We can use the `RND` effect to implement a simple guessing game. The `guess` function, given a target number, will repeatedly ask the user for a guess, and state whether the guess is too high, too low, or correct:

```
guess : Int -> Eff () [STDIO]
```

For reference, the code for `guess` is given below:

```
guess : Int -> Eff () [STDIO]
guess target
  = do putStr "Guess: "
      case run {m=Maybe} (parseNumber 100 (trim !getStr)) of
        Nothing => do putStrLn "Invalid input"
                     guess target
        Just (v ** _) =>
          case compare v target of
            LT => do putStrLn "Too low"
                  guess target
            EQ => putStrLn "You win!"
            GT => do putStrLn "Too high"
                  guess target
```

Note that we use `parseNumber` as defined previously to read user input, but we don't need to list the `EXCEPTION` effect because we use a nested `run` to invoke `parseNumber`, independently of the calling effectful program.

To invoke this, we pick a random number within the range 0–100, having set up the random number generator with a seed, then run `guess`:

```
game : Eff () [RND, STDIO]
game = do srand 123456789
         guess (fromInteger !(rndInt 0 100))

main : IO ()
main = run game
```

If no seed is given, it is set to the `default` value. For a less predictable game, some better source of randomness would be required, for example taking an initial seed from the system time. To see how to do this, see the `SYSTEM` effect described in *Effects Summary*.

## 3.4 Non-determinism

The listing below gives the definition of the non-determinism effect, which allows a program to choose a value non-deterministically from a list of possibilities in such a way that the entire computation succeeds:

```
import Effects
import Effect.Select

SELECT : EFFECT

select : List a -> Eff a [SELECT]

instance Handler Selection Maybe
instance Handler Selection List
```

### 3.4.1 Example

The `SELECT` effect can be used to solve constraint problems, such as finding Pythagorean triples. The idea is to use `select` to give a set of candidate values, then throw an exception for any combination of values which does not satisfy the constraint:

```
triple : Int -> Eff (Int, Int, Int) [SELECT, EXCEPTION String]
triple max = do z <- select [1..max]
               y <- select [1..z]
               x <- select [1..y]
               if (x * x + y * y == z * z)
                 then pure (x, y, z)
                 else raise "No triple"
```

This program chooses a value for `z` between 1 and `max`, then values for `y` and `x`. In operation, after a `select`, the program executes the rest of the `do`-block for every possible assignment, effectively searching depth-first. If the list is empty (or an exception is thrown) execution fails.

There are handlers defined for `Maybe` and `List` contexts, i.e. contexts which can capture failure. Depending on the context `m`, `triple` will either return the first triple it finds (if in `Maybe` context) or all triples in the range (if in `List` context). We can try this as follows:

```
main : IO ()
main = do print $ the (Maybe _) $ run (triple 100)
         print $ the (List _) $ run (triple 100)
```

## 3.5 vadd revisited

We now return to the `vadd` program from the introduction. Recall the definition:

```
vadd : Vect n Int -> Vect n Int -> Vect n Int
vadd [] [] = []
vadd (x :: xs) (y :: ys) = x + y :: vadd xs ys
```

Using `,` we can set up a program so that it reads input from a user, checks that the input is valid (i.e both vectors contain integers, and are the same length) and if so, pass it on to `vadd`. First, we write a wrapper for `vadd` which checks the lengths and throw an exception if they are not equal. We can do this for input vectors of length `n` and `m` by matching on the implicit arguments `n` and `m` and using `decEq` to produce a proof of their equality, if they are equal:

```
vadd_check : Vect n Int -> Vect m Int ->
  Eff (Vect m Int) [EXCEPTION String]
vadd_check {n} {m} xs ys with (decEq n m)
  vadd_check {n} {m=n} xs ys | (Yes Refl) = pure (vadd xs ys)
  vadd_check {n} {m} xs ys | (No contra) = raise "Length mismatch"
```

To read a vector from the console, we implement a function of the following type:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
```

This returns a dependent pair of a length, and a vector of that length, because we cannot know in advance how many integers the user is going to input. We can use `-1` to indicate the end of input:

```
read_vec : Eff (p ** Vect p Int) [STDIO]
read_vec = do putStr "Number (-1 when done): "
  case the (Maybe Int) $ run (parseNumber (trim !getStr)) of
    Nothing => do putStrLn "Input error"
                  read_vec
    Just v => if (v /= -1)
      then do (_ ** xs) <- read_vec
              pure (_ ** v :: xs)
      else pure (_ ** [])

where
  parseNumber : String -> Eff Int [EXCEPTION String]
  parseNumber str
    = if all (\x => isDigit x || x == '-') (unpack str)
      then pure (cast str)
      else raise "Not a number"
```

This uses a variation on `parseNumber` which does not require a number to be within range.

Finally, we write a program which reads two vectors and prints the result of pairwise addition of them, throwing an exception if the inputs are of differing lengths:

```
do_vadd : Eff () [STDIO, EXCEPTION String]
do_vadd = do putStrLn "Vector 1"
  (_ ** xs) <- read_vec
  putStrLn "Vector 2"
  (_ ** ys) <- read_vec
  putStrLn (show !(vadd_check xs ys))
```

By having explicit lengths in the type, we can be sure that `vadd` is only being used where the lengths of inputs are guaranteed to be equal. This does not stop us reading vectors from user input, but it does require that the lengths are checked and any discrepancy is dealt with gracefully.

### 3.6 Example: An Expression Calculator

To show how these effects can fit together, let us consider an evaluator for a simple expression language, with addition and integer values.

```
data Expr = Val Integer
          | Add Expr Expr
```

An evaluator for this language always returns an `Integer`, and there are no situations in which it can fail!

```
eval : Expr -> Integer
eval (Val x) = x
eval (Add l r) = eval l + eval r
```

If we add variables, however, things get more interesting. The evaluator will need to be able to access the values stored in variables, and variables may be undefined.

```
data Expr = Val Integer
         | Var String
         | Add Expr Expr
```

To start, we will change the type of `eval` so that it is effectful, and supports an exception effect for throwing errors, and a state containing a mapping from variable names (as `String`) to their values:

```
Env : Type
Env = List (String, Integer)

eval : Expr -> Eff Integer [EXCEPTION String, STATE Env]
eval (Val x) = return x
eval (Add l r) = return $ !(eval l) + !(eval r)
```

Note that we are using `!`-notation to avoid having to bind subexpressions in a `do` block. Next, we add a case for evaluating `Var`:

```
eval (Var x) = case lookup x !get of
               Nothing => raise $ "No such variable " ++ x
               Just val => return val
```

This retrieves the state (with `get`, supported by the `STATE Env` effect) and raises an exception if the variable is not in the environment (with `raise`, supported by the `EXCEPTION String` effect).

To run the evaluator on a particular expression in a particular environment of names and their values, we can write a function which sets the state then invokes `eval`:

```
runEval : List (String, Integer) -> Expr -> Maybe Integer
runEval args expr = run (eval' expr)
  where eval' : Expr -> Eff Integer [EXCEPTION String, STATE Env]
        eval' e = do put args
                    eval e
```

We have picked `Maybe` as a computation context here; it needs to be a context which is available for every effect supported by `eval`. In particular, because we have exceptions, it needs to be a context which supports exceptions. Alternatively, `Either String` or `IO` would be fine, for example.

What if we want to extend the evaluator further, with random number generation? To achieve this, we add a new constructor to `Expr`, which gives a random number up to a maximum value:

```
data Expr = Val Integer
         | Var String
         | Add Expr Expr
         | Random Integer
```

Then, we need to deal with the new case, making sure that we extend the list of events to include `RND`. It doesn't matter where `RND` appears in the list, as long as it is present:

```
eval : Expr -> Eff Integer [EXCEPTION String, RND, STATE Env]

eval (Random upper) = rndInt 0 upper
```

For test purposes, we might also want to print the random number which has been generated:

```
eval (Random upper) = do val <- rndInt 0 upper
                        putStrLn (show val)
                        return val
```

If we try this without extending the effects list, we would see an error something like the following:

```
Expr.idr:28:6:When elaborating right hand side of eval:
Can't solve goal
  SubList [STDIO]
    [(EXCEPTION String), RND, (STATE (List (String, Integer)))]
```

In other words, the `STDIO` effect is not available. We can correct this simply by updating the type of `eval` to include `STDIO`.

```
eval : Expr -> Eff Integer [STDIO, EXCEPTION String, RND, STATE Env]
```

Note that using `STDIO` will restrict the number of contexts in which `eval` can be run to those which support `STDIO`, such as `IO`. Once effect lists get longer, it can be a good idea instead to encapsulate sets of effects in a type synonym. This is achieved as follows, simply by defining a function which computes a type, since types are first class in Idris:

```
EvalEff : Type -> Type
EvalEff t = Eff t [STDIO, EXCEPTION String, RND, STATE Env]

eval : Expr -> EvalEff Integer
```



---

## Dependent Effects

---

In the programs we have seen so far, the available effects have remained constant. Sometimes, however, an operation can *change* the available effects. The simplest example occurs when we have a state with a dependent type—adding an element to a vector also changes its type, for example, since its length is explicit in the type. In this section, we will see how the library supports this. Firstly, we will see how states with dependent types can be implemented. Secondly, we will see how the effects can depend on the *result* of an effectful operation. Finally, we will see how this can be used to implement a type-safe and resource-safe protocol for file management.

### 4.1 Dependent States

Suppose we have a function which reads input from the console, converts it to an integer, and adds it to a list which is stored in a STATE. It might look something like the following:

```
readInt : Eff () [STATE (List Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

But what if, instead of a list of integers, we would like to store a `Vect`, maintaining the length in the type?

```
readInt : Eff () [STATE (Vect n Int), STDIO]
readInt = do let x = trim !getStr
             put (cast x :: !get)
```

This will not type check! Although the vector has length `n` on entry to `readInt`, it has length `S n` on exit. The library allows us to express this as follows:

```
readInt : Eff () [STATE (Vect n Int), STDIO]
           [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
             putM (cast x :: !get)
```

The type `Eff a xs xs'` means that the operation begins with effects `xs` available, and ends with effects `xs'` available. We have used `putM` to update the state, where the `M` suffix indicates that the *type* is being updated as well as the value. It has the following type:

```
putM : y -> Eff () [STATE x] [STATE y]
```

### 4.2 Result-dependent Effects

Often, whether a state is updated could depend on the success or otherwise of an operation. In our `readInt` example, we might wish to update the vector only if the input is a valid integer (i.e. all digits). As a first attempt, we could try the following, returning a `Bool` which indicates success:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
           [STATE (Vect (S n) Int), STDIO]
readInt = do let x = trim !getStr
              case all isDigit (unpack x) of
                False => pure False
                True  => do putM (cast x :: !get)
                           pure True
```

Unfortunately, this will not type check because the vector does not get extended in both branches of the case!

```
MutState.idr:18:19:When elaborating right hand side of Main.case
block in readInt:
Unifying n and S n would lead to infinite value
```

Clearly, the size of the resulting vector depends on whether or not the value read from the user was valid. We can express this in the type:

```
readInt : Eff Bool [STATE (Vect n Int), STDIO]
           (\ok => if ok then [STATE (Vect (S n) Int), STDIO]
                    else [STATE (Vect n Int), STDIO])
readInt = do let x = trim !getStr
              case all isDigit (unpack x) of
                False => pureM False
                True  => do putM (cast x :: !get)
                           pureM True
```

Using `pureM` rather than `pure` allows the output effects to be calculated from the value given. Its type is:

```
pureM : (val : a) -> EffM m a (xs val) xs
```

When using `readInt`, we will have to check its return value in order to know what the new set of effects is. For example, to read a set number of values into a vector, we could write the following:

```
readN : (n : Nat) ->
        Eff () [STATE (Vect m Int), STDIO]
              [STATE (Vect (n + m) Int), STDIO]
readN Z = pure ()
readN {m} (S k) = case !readInt of
                    True => rewrite plusSuccRightSucc k m in readN k
                    False => readN (S k)
```

The case analysis on the result of `readInt` means that we know in each branch whether reading the integer succeeded, and therefore how many values still need to be read into the vector. What this means in practice is that the type system has verified that a necessary dynamic check (i.e. whether reading a value succeeded) has indeed been done.

---

**Note:** Only case will work here. We cannot use `if/then/else` because the `then` and `else` branches must have the same type. The `case` construct, however, abstracts over the value being inspected in the type of each branch.

---

## 4.3 File Management

A practical use for dependent effects is in specifying resource usage protocols and verifying that they are executed correctly. For example, file management follows a resource usage protocol with the following (informally specified) requirements:

- It is necessary to open a file for reading before reading it
- Opening may fail, so the programmer should check whether opening was successful
- A file which is open for reading must not be written to, and vice versa

- When finished, an open file handle should be closed
- When a file is closed, its handle should no longer be used

These requirements can be expressed formally in `Effect`, by creating a `FILE_IO` effect parameterised over a file handle state, which is either empty, open for reading, or open for writing. The `FILE_IO` effect's definition is given below. Note that this effect is mainly for illustrative purposes—typically we would also like to support random access files and better reporting of error conditions.

```

module Effect.File

import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
      -> (m : Mode)
      -> Eff Bool [FILE_IO ()]
      (\res => [FILE_IO (case res of
                    True => OpenFile m
                    False => ())])

close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

instance Handler FileIO IO

```

In particular, consider the type of `open`:

```

open : (fname : String)
      -> (m : Mode)
      -> Eff Bool [FILE_IO ()]
      (\res => [FILE_IO (case res of
                    True => OpenFile m
                    False => ())])

```

This returns a `Bool` which indicates whether opening the file was successful. The resulting state depends on whether the operation was successful; if so, we have a file handle open for the stated purpose, and if not, we have no file handle. By `case` analysis on the result, we continue the protocol accordingly.

```

readFile : Eff (List String) [FILE_IO (OpenFile Read)]
readFile = readAcc [] where
  readAcc : List String -> Eff (List String) [FILE_IO (OpenFile Read)]
  readAcc acc = if (not !eof)
                 then readAcc (!readLine :: acc)
                 else pure (reverse acc)

```

Given a function `readFile`, above, which reads from an open file until reaching the end, we can write a program which opens a file, reads it, then displays the contents and closes it, as follows, correctly following the protocol:

```

dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = case !(open name Read) of
  True => do putStrLn (show !readFile)
            close
  False => putStrLn ("Error!")

```

The type of `dumpFile`, with `FILE_IO ()` in its effect list, indicates that any use of the file resource will follow the protocol correctly (i.e. it both begins and ends with an empty resource). If we fail to follow the protocol correctly (perhaps by forgetting to close the file, failing to check that `open` succeeded, or opening the file for

writing) then we will get a compile-time error. For example, changing open name Read to open name Write yields a compile-time error of the following form:

```
FileTest.idr:16:18:When elaborating right hand side of Main.case
block in testFile:
Can't solve goal
    SubList [(FILE_IO (OpenFile Read))]
            [(FILE_IO (OpenFile Write)), STDIO]
```

In other words: when reading a file, we need a file which is open for reading, but the effect list contains a FILE\_IO effect carrying a file open for writing.

## 4.4 Pattern-matching bind

It might seem that having to test each potentially failing operation with a case clause could lead to ugly code, with lots of nested case blocks. Many languages support exceptions to improve this, but unfortunately exceptions may not allow completely clean resource management—for example, guaranteeing that any open which did succeed has a corresponding close.

Idris supports *pattern-matching* bindings, such as the following:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = do True <- open name Read
                 putStrLn (show !readFile)
                 close
```

This also has a problem: we are no longer dealing with the case where opening a file failed! The solution is to extend the pattern-matching binding syntax to give brief clauses for failing matches. Here, for example, we could write:

```
dumpFile : String -> Eff () [FILE_IO (), STDIO]
dumpFile name = do True <- open name Read | False => putStrLn "Error"
                 putStrLn (show !readFile)
                 close
```

This is exactly equivalent to the definition with the explicit case. In general, in a do-block, the syntax:

```
do pat <- val | <alternatives>
  p
```

is desugared to

```
do x <- val
  case x of
    pat => p
    <alternatives>
```

There can be several alternatives, separated by a vertical bar |. For example, there is a SYSTEM effect which supports reading command line arguments, among other things (see Appendix *Effects Summary*). To read command line arguments, we can use getArgs:

```
getArgs : Eff (List String) [SYSTEM]
```

A main program can read command line arguments as follows, where in the list which is returned, the first element prog is the executable name and the second is an expected argument:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs
          putStrLn $ "Argument is " ++ arg
          {- ... rest of function ... -}
```

Unfortunately, this will not fail gracefully if no argument is given, or if too many arguments are given. We can use pattern matching bind alternatives to give a better (more informative) error:

```
emain : Eff () [SYSTEM, STDIO]
emain = do [prog, arg] <- getArgs | [] => putStrLn "Can't happen!"
          | [prog] => putStrLn "No arguments!"
          | _ => putStrLn "Too many arguments!"
          putStrLn $ "Argument is " ++ arg
          {- ... rest of function ... -}
```

If `getArgs` does not return something of the form `[prog, arg]` the alternative which does match is executed instead, and that value returned.



---

## Creating New Effects

---

We have now seen several side-effecting operations provided by the `Effects` library, and examples of their use in Section *Simple Effects*. We have also seen how operations may *modify* the available effects by changing state in Section *Dependent Effects*. We have not, however, yet seen how these operations are implemented. In this section, we describe how a selection of the available effects are implemented, and show how new effectful operations may be provided.

### 5.1 State

Effects are described by *algebraic data types*, where the constructors describe the operations provided when the effect is available. Stateful operations are described as follows:

```
data State : Effect where
  Get :      State a a (\x => a)
  Put : b -> State () a (\x => b)
```

`Effect` itself is a type synonym, giving the required type for an effect signature:

```
Effect : Type
Effect = (result : Type) ->
         (input_resource : Type) ->
         (output_resource : result -> Type) -> Type
```

Each effect is associated with a *resource*. The second argument to an effect signature is the resource type on *input* to an operation, and the third is a function which computes the resource type on *output*. Here, it means:

- `Get` takes no arguments. It has a resource of type `a`, which is not updated, and running the `Get` operation returns something of type `a`.
- `Put` takes a `b` as an argument. It has a resource of type `a` on input, which is updated to a resource of type `b`. Running the `Put` operation returns the element of the unit type.

The effects library provides an overloaded function `sig` which can make effect signatures more concise, particularly when the result has no effect on the resource type. For `State`, we can write:

```
data State : Effect where
  Get :      sig State a a
  Put : b -> sig State () a b
```

There are four versions of `sig`, depending on whether we are interested in the resource type, and whether we are updating the resource. Idris will infer the appropriate version from usage.

```
NoResourceEffect.sig : Effect -> Type -> Type
NoUpdateEffect.sig   : Effect -> (ret : Type) ->
                           (resource : Type) -> Type
UpdateEffect.sig     : Effect -> (ret : Type) ->
                           (resource_in : Type) ->
```

```

DepUpdateEffect.sig : Effect -> (ret : Type) ->
                          (resource_in : Type) ->
                          (resource_out : ret -> Type) -> Type

```

In order to convert `State` (of type `Effect`) into something usable in an effects list, of type `EFFECT`, we write the following:

```

STATE : Type -> EFFECT
STATE t = MkEff t State

```

`MkEff` constructs an `EFFECT` by taking the resource type (here, the `t` which parameterises `STATE`) and the effect signature (here, `State`). For reference, `EFFECT` is declared as follows:

```

data EFFECT : Type where
  MkEff : Type -> Effect -> EFFECT

```

Recall that to run an effectful program in `Eff`, we use one of the `run` family of functions to run the program in a particular computation context `m`. For each effect, therefore, we must explain how it is executed in a particular computation context for `run` to work in that context. This is achieved with the following type class:

```

class Handler (e : Effect) (m : Type -> Type) where
  handle : resource -> (eff : e t resource resource') ->
    ((x : t) -> resource' x -> m a) -> m a

```

We have already seen some instance declarations in the effect summaries in Section *Simple Effects*. An instance of `Handler e m` means that the effect declared with signature `e` can be run in computation context `m`. The `handle` function takes:

- The `resource` on input (so, the current value of the state for `State`)
- The effectful operation (either `Get` or `Put x` for `State`)
- A *continuation*, which we conventionally call `k`, and should be passed the result value of the operation, and an updated resource.

There are two reasons for taking a continuation here: firstly, this is neater because there are multiple return values (a new resource and the result of the operation); secondly, and more importantly, the continuation can be called zero or more times.

A `Handler` for `State` simply passes on the value of the state, in the case of `Get`, or passes on a new state, in the case of `Put`. It is defined the same way for all computation contexts:

```

instance Handler State m where
  handle st Get      k = k st st
  handle st (Put n) k = k () n

```

This gives enough information for `Get` and `Put` to be used directly in `Eff` programs. It is tidy, however, to define top level functions in `Eff`, as follows:

```

get : Eff x [STATE x]
get = call Get

put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)

```

**An implementation detail (aside):** The `call` function converts an `Effect` to a function in `Eff`, given a proof that the effect is available. This proof can be constructed automatically by `by`, since it is essentially an index into a statically known list of effects:

```

call : {e : Effect} ->
  (eff : e t a b) -> {auto prf : EffElem e a xs} ->
  Eff t xs (\v => updateResTy v xs prf eff)

```



This is the reason for the `Can't solve goal error` when an effect is not available: the implicit proof `prf` has not been solved automatically because the required effect is not in the list of effects `xs`.

Such details are not important for using the library, or even writing new effects, however.

### 5.1.1 Summary

The following listing summarises what is required to define the `STATE` effect:

```
data State : Effect where
  Get : sig State a a
  Put : b -> sig State () a b

STATE : Type -> EFFECT
STATE t = MkEff t State

instance Handler State m where
  handle st Get k = k st st
  handle st (Put n) k = k () n

get : Eff x [STATE x]
get = call Get

put : x -> Eff () [STATE x]
put val = call (Put val)

putM : y -> Eff () [STATE x] [STATE y]
putM val = call (Put val)
```

## 5.2 Console I/O

Then listing below gives the definition of the `STDIO` effect, including handlers for `IO` and `IOExcept`. We omit the definition of the top level `Eff` functions, as this merely invoke the effects `PutStr`, `GetStr`, `PutCh` and `GetCh` directly.

Note that in this case, the resource is the unit type in every case, since the handlers merely apply the `IO` equivalents of the effects directly.

```
data StdIO : Effect where
  PutStr : String -> sig StdIO ()
  GetStr : sig StdIO String
  PutCh : Char -> sig StdIO ()
  GetCh : sig StdIO Char

instance Handler StdIO IO where
  handle () (PutStr s) k = do putStr s; k () ()
  handle () GetStr k = do x <- getLine; k x ()
  handle () (PutCh c) k = do putChar c; k () ()
  handle () GetCh k = do x <- getChar; k x ()

instance Handler StdIO (IOExcept a) where
  handle () (PutStr s) k = do ioe_lift $ putStr s; k () ()
  handle () GetStr k = do x <- ioe_lift $ getLine; k x ()
  handle () (PutCh c) k = do ioe_lift $ putChar c; k () ()
  handle () GetCh k = do x <- ioe_lift $ getChar; k x ()

STDIO : EFFECT
STDIO = MkEff () StdIO
```

## 5.3 Exceptions

The listing below gives the definition of the `Exception` effect, including two of its handlers for `Maybe` and `List`. The only operation provided is `Raise`. The key point to note in the definitions of these handlers is that the continuation `k` is not used. Running `Raise` therefore means that computation stops with an error.

```
data Exception : Type -> Effect where
  Raise : a -> sig (Exception a) b

instance Handler (Exception a) Maybe where
  handle _ (Raise e) k = Nothing

instance Handler (Exception a) List where
  handle _ (Raise e) k = []

EXCEPTION : Type -> EFFECT
EXCEPTION t = MkEff () (Exception t)
```

## 5.4 Non-determinism

The following listing gives the definition of the `Select` effect for writing non-deterministic programs, including a handler for `List` context which returns all possible successful values, and a handler for `Maybe` context which returns the first successful value.

```
data Selection : Effect where
  Select : List a -> sig Selection a

instance Handler Selection Maybe where
  handle _ (Select xs) k = tryAll xs where
    tryAll [] = Nothing
    tryAll (x :: xs) = case k x () of
      Nothing => tryAll xs
      Just v => Just v

instance Handler Selection List where
  handle r (Select xs) k = concatMap (\x => k x r) xs

SELECT : EFFECT
SELECT = MkEff () Selection
```

Here, the continuation is called multiple times in each handler, for each value in the list of possible values. In the `List` handler, we accumulate all successful results, and in the `Maybe` handler we try the first value in the list, and try later values only if that fails.

## 5.5 File Management

Result-dependent effects are no different from non-dependent effects in the way they are implemented. The listing below illustrates this for the `FILE_IO` effect. The syntax for state transitions `{ x ==> {res} x' }`, where the result state `x'` is computed from the result of the operation `res`, follows that for the equivalent `Eff` programs.

```
data FileIO : Effect where
  Open : (fname: String)
        -> (m : Mode)
        -> sig FileIO Bool () (\res => case res of
          True => OpenFile m
          False => ())

  Close : sig FileIO () (OpenFile m)
```

```

ReadLine  :          sig FileIO String (OpenFile Read)
WriteLine : String -> sig FileIO ()    (OpenFile Write)
EOF       :          sig FileIO Bool   (OpenFile Read)

instance Handler FileIO IO where
  handle () (Open fname m) k = do h <- openFile fname m
                                if !(validFile h)
                                  then k True (FH h)
                                  else k False ()

  handle (FH h) Close      k = do closeFile h
                                k () ()

  handle (FH h) ReadLine   k = do str <- fread h
                                k str (FH h)

  handle (FH h) (WriteLine str) k = do fwrite h str
                                k () (FH h)

  handle (FH h) EOF        k = do e <- feof h
                                k e (FH h)

FILE_IO : Type -> EFFECT
FILE_IO t = MkEff t FileIO

```

Note that in the handler for `Open`, the types passed to the continuation `k` are different depending on whether the result is `True` (opening succeeded) or `False` (opening failed). This uses `validFile`, defined in the `Prelude`, to test whether a file handler refers to an open file or not.



---

## Example: A “Mystery Word” Guessing Game

---

In this section, we will use the techniques and specific effects discussed in the tutorial so far to implement a larger example, a simple text-based word-guessing game. In the game, the computer chooses a word, which the player must guess letter by letter. After a limited number of wrong guesses, the player loses <sup>1</sup>.

We will implement the game by following these steps:

1. Define the game state, in enough detail to express the rules
2. Define the rules of the game (i.e. what actions the player may take, and how these actions affect the game state)
3. Implement the rules of the game (i.e. implement state updates for each action)
4. Implement a user interface which allows a player to direct actions

Step 2 may be achieved by defining an effect which depends on the state defined in step 1. Then step 3 involves implementing a `Handler` for this effect. Finally, step 4 involves implementing a program in `Eff` using the newly defined effect (and any others required to implement the interface).

### 6.1 Step 1: Game State

First, we categorise the game states as running games (where there are a number of guesses available, and a number of letters still to guess), or non-running games (i.e. games which have not been started, or games which have been won or lost).

```
data GState = Running Nat Nat | NotRunning
```

Notice that at this stage, we say nothing about what it means to make a guess, what the word to be guessed is, how to guess letters, or any other implementation detail. We are only interested in what is necessary to describe the game rules.

We will, however, parameterise a concrete game state `Mystery` over this data:

```
data Mystery : GState -> Type
```

### 6.2 Step 2: Game Rules

We describe the game rules as a dependent effect, where each action has a *precondition* (i.e. what the game state must be before carrying out the action) and a *postcondition* (i.e. how the action affects the game state). Informally, these actions with the pre- and postconditions are:

**Guess** Guess a letter in the word.

<sup>1</sup> Readers may recognise this game by the name “Hangman”.

- Precondition: The game must be running, and there must be both guesses still available, and letters still to be guessed.
- Postcondition: If the guessed letter is in the word and not yet guessed, reduce the number of letters, otherwise reduce the number of guesses.

**Won** Declare victory

- Precondition: The game must be running, and there must be no letters still to be guessed.
- Postcondition: The game is no longer running.

**Lost** Accept defeat

- Precondition: The game must be running, and there must be no guesses left.
- Postcondition: The game is no longer running.

**NewWord** Set a new word to be guessed

- Precondition: The game must not be running.
- Postcondition: The game is running, with 6 guesses available (the choice of 6 is somewhat arbitrary here) and the number of unique letters in the word still to be guessed.

**Get** Get a string representation of the game state. This is for display purposes; there are no pre- or postconditions.

We can make these rules precise by declaring them more formally in an effect signature:

```
data MysteryRules : Effect where
  Guess : (x : Char) ->
    sig MysteryRules Bool
      (Mystery (Running (S g) (S w)))
      (\inword => if inword
                  then Mystery (Running (S g) w)
                  else Mystery (Running g (S w)))
  Won : sig MysteryRules () (Mystery (Running g 0))
      (Mystery NotRunning)
  Lost : sig MysteryRules () (Mystery (Running 0 g))
      (Mystery NotRunning)
  NewWord : (w : String) ->
    sig MysteryRules () (Mystery NotRunning) (Mystery (Running 6 (length (letters w))))
  Get : sig MysteryRules String (Mystery h)
```

This description says nothing about how the rules are implemented. In particular, it does not specify *how* to tell whether a guessed letter was in a word, just that the result of `Guess` depends on it.

Nevertheless, we can still create an EFFECT from this, and use it in an `Eff` program. Implementing a `Handler` for `MysteryRules` will then allow us to play the game.

```
MYSTERY : GState -> EFFECT
MYSTERY h = Mkeff (Mystery h) MysteryRules
```

## 6.3 Step 3: Implement Rules

To *implement* the rules, we begin by giving a concrete definition of game state:

```
data Mystery : GState -> Type where
  Init : Mystery NotRunning
  GameWon : (word : String) -> Mystery NotRunning
  GameLost : (word : String) -> Mystery NotRunning
  MkG : (word : String) ->
    (guesses : Nat) ->
    (got : List Char) ->
    (missing : Vect m Char) ->
    Mystery (Running guesses m)
```

If a game is `NotRunning`, that is either because it has not yet started (`Init`) or because it is won or lost (`GameWon` and `GameLost`, each of which carry the word so that showing the game state will reveal the word to the player). Finally, `MkG` captures a running game's state, including the target word, the letters successfully guessed, and the missing letters. Using a `Vect` for the missing letters is convenient since its length is used in the type.

To initialise the state, we implement the following functions: `letters`, which returns a list of unique letters in a `String` (ignoring spaces) and `initState` which sets up an initial state considered valid as a postcondition for `NewWord`.

```
letters : String -> List Char
initState : (x : String) -> Mystery (Running 6 (length (letters x)))
```

When checking if a guess is in the vector of missing letters, it is convenient to return a *proof* that the guess is in the vector, using `isElem` below, rather than merely a `Bool`:

```
data IsElem : a -> Vect n a -> Type where
  First : IsElem x (x :: xs)
  Later : IsElem x xs -> IsElem x (y :: xs)

isElem : DecEq a => (x : a) -> (xs : Vect n a) -> Maybe (IsElem x xs)
```

The reason for returning a proof is that we can use it to remove an element from the correct position in a vector:

```
shrink : (xs : Vect (S n) a) -> IsElem x xs -> Vect n a
```

We leave the definitions of `letters`, `init`, `isElem` and `shrink` as exercises. Having implemented these, the `Handler` implementation for `MysteryRules` is surprisingly straightforward:

```
instance Handler MysteryRules m where
  handle (MkG w g got []) Won k = k () (GameWon w)
  handle (MkG w Z got m) Lost k = k () (GameLost w)

  handle st Get k = k (show st) st
  handle st (NewWord w) k = k () (initState w)

  handle (MkG w (S g) got m) (Guess x) k =
    case isElem x m of
      Nothing => k False (MkG w _ got m)
      (Just p) => k True (MkG w _ (x :: got) (shrink m p))
```

Each case simply involves directly updating the game state in a way which is consistent with the declared rules. In particular, in `Guess`, if the handler claims that the guessed letter is in the word (by passing `True` to `k`), there is no way to update the state in such a way that the number of missing letters or number of guesses does not follow the rules.

## 6.4 Step 4: Implement Interface

Having described the rules, and implemented state transitions which follow those rules as an effect handler, we can now write an interface for the game which uses the `MYSTERY` effect:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
         [MYSTERY NotRunning, STDIO]
```

The type indicates that the game must start in a running state, with some guesses available, and eventually reach a not-running state (i.e. won or lost). The only way to achieve this is by correctly following the stated rules.

Note that the type of `game` makes no assumption that there are letters to be guessed in the given word (i.e. it is `w` rather than `S w`). This is because we will be choosing a word at random from a vector of `String`, and at no point have we made it explicit that those `String` are non-empty.

Finally, we need to initialise the game by picking a word at random from a list of candidates, setting it as the target using `NewWord`, then running `game`:

```
runGame : Eff () [MYSTERY NotRunning, RND, SYSTEM, STDIO]
runGame = do srand !time
           let w = index !(rndFin _) words
           call $ NewWord w
           game
           putStrLn !(call Get)
```

We use the system time (`time` from the `SYSTEM` effect; see Appendix *Effects Summary*) to initialise the random number generator, then pick a random `Fin` to index into a list of `words`. For example, we could initialise a word list as follows:

```
words : ?wtype
words = with Vect ["idris", "agda", "haskell", "miranda",
                  "java", "javascript", "fortran", "basic",
                  "coffeescript", "rust"]

wtype = proof search
```

**Note:** Rather than have to explicitly declare a type with the vector's length, it is convenient to give a hole `?wtype` and let Idris's proof search mechanism find the type. This is a limited form of type inference, but very useful in practice.

A possible complete implementation of `game` is presented below:

```
game : Eff () [MYSTERY (Running (S g) w), STDIO]
           [MYSTERY NotRunning, STDIO]
game {w=Z} = Won
game {w=S _}
  = do putStrLn !Get
       putStr "Enter guess: "
       let guess = trim !getStr
           case choose (not (guess == "")) of
             (Left p) => processGuess (strHead' guess p)
             (Right p) => do putStrLn "Invalid input!"
                           game

where
  processGuess : Char -> Eff () [MYSTERY (Running (S g) (S w)), STDIO]
                    [MYSTERY NotRunning, STDIO]
  processGuess {g} {w} c
    = case !(Main.Guess c) of
      True => do putStrLn "Good guess!"
                case w of
                  Z => Won
                  (S k) => game
      False => do putStrLn "No, sorry"
                 case g of
                   Z => Lost
                   (S k) => game
```

## 6.5 Discussion

Writing the rules separately as an effect, then an implementation which uses that effect, ensures that the implementation must follow the rules. This has practical applications in more serious contexts; `MysteryRules` for example can be thought of as describing a *protocol* that a game player must follow, or alternative a *precisely-typed API*.

In practice, we wouldn't really expect to write rules first then implement the game once the rules were complete. Indeed, I didn't do so when constructing this example! Rather, I wrote down a set of likely rules making any assumptions *explicit* in the state transitions for `MysteryRules`. Then, when implementing `game` at first, any incorrect assumption was caught as a type error. The following errors were caught during development:



- Not realising that allowing `NewWord` to be an arbitrary string would mean that `game` would have to deal with a zero-length word as a starting state.
- Forgetting to check whether a game was won before recursively calling `processGuess`, thus accidentally continuing a finished game.
- Accidentally checking the number of missing letters, rather than the number of remaining guesses, when checking if a game was lost.

These are, of course, simple errors, but were caught by the type checker before any testing of the game.



---

## Further Reading

---

This tutorial has given an introduction to writing and reasoning about side-effecting programs in Idris, using the `Effects` library. More details about the *implementation* of the library, such as how `run` works, how handlers are invoked, etc, are given in a separate paper <sup>1</sup>.

Some libraries and programs which use `Effects` can be found in the following places:

- <https://github.com/edwinb/SDL-idris> — some bindings for the SDL media library, supporting graphics in particular.
- <https://github.com/edwinb/idris-demos> — various demonstration programs, including several examples from this tutorial, and a “Space Invaders” game.
- <https://github.com/SimonJF/IdrisNet2> — networking and socket libraries.
- <https://github.com/edwinb/Protocols> — a high level communication protocol description language.

The inspiration for the `Effects` library was Bauer and Pretnar’s `Eff` language <sup>2</sup>, which describes a language based on algebraic effects and handlers. Other recent languages and libraries have also been built on this ideas, for example <sup>3</sup> and <sup>4</sup>. The theoretical foundations are also well-studied see <sup>5</sup>, <sup>6</sup>, <sup>7</sup>, <sup>8</sup>.

---

<sup>1</sup> Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. SIGPLAN Not. 48, 9 (September 2013), 133-144. DOI=10.1145/2544174.2500581 <https://dl.acm.org/citation.cfm?doid=2544174.2500581>

<sup>2</sup> Andrej Bauer, Matija Pretnar, Programming with algebraic effects and handlers, Journal of Logical and Algebraic Methods in Programming, Volume 84, Issue 1, January 2015, Pages 108-123, ISSN 2352-2208, <http://math.andrej.com/wp-content/uploads/2012/03/eff.pdf>

<sup>3</sup> Ben Lippmeier. 2009. Witnessing Purity, Constancy and Mutability. In Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS '09), Zhenjiang Hu (Ed.). Springer-Verlag, Berlin, Heidelberg, 95-110. DOI=10.1007/978-3-642-10672-9\_9 [http://link.springer.com/chapter/10.1007%2F978-3-642-10672-9\\_9](http://link.springer.com/chapter/10.1007%2F978-3-642-10672-9_9)

<sup>4</sup> Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. SIGPLAN Not. 48, 9 (September 2013), 145-158. DOI=10.1145/2544174.2500590 <https://dl.acm.org/citation.cfm?doid=2544174.2500590>

<sup>5</sup> Martin Hyland, Gordon Plotkin, John Power, Combining effects: Sum and tensor, Theoretical Computer Science, Volume 357, Issues 1-3, 25 July 2006, Pages 70-99, ISSN 0304-3975, (<https://www.sciencedirect.com/science/article/pii/S0304397506002659>)

<sup>6</sup> Paul Blain Levy. 2004. Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2). Kluwer Academic Publishers, Norwell, MA, USA.

<sup>7</sup> Plotkin, Gordon, and Matija Pretnar. “Handlers of algebraic effects.” Programming Languages and Systems. Springer Berlin Heidelberg, 2009. 80-94.

<sup>8</sup> Pretnar, Matija. “Logic and handling of algebraic effects.” (2010).



---

## Effects Summary

---

This appendix gives interfaces for the core effects provided by the library.

### 8.1 EXCEPTION

```

module Effect.Exception

import Effects
import System
import Control.IOExcept

EXCEPTION : Type -> EFFECT

raise : a -> Eff b [EXCEPTION a]

instance          Handler (Exception a) Maybe
instance          Handler (Exception a) List
instance          Handler (Exception a) (Either a)
instance          Handler (Exception a) (IOExcept a)
instance Show a => Handler (Exception a) IO

```

### 8.2 FILE\_IO

```

module Effect.File

import Effects
import Control.IOExcept

FILE_IO : Type -> EFFECT

data OpenFile : Mode -> Type

open : (fname : String)
      -> (m : Mode)
      -> Eff Bool [FILE_IO ()]
      (\res => [FILE_IO (case res of
                    True => OpenFile m
                    False => ())])

close : Eff () [FILE_IO (OpenFile m)] [FILE_IO ()]

readLine : Eff String [FILE_IO (OpenFile Read)]
writeLine : String -> Eff () [FILE_IO (OpenFile Write)]
eof       : Eff Bool [FILE_IO (OpenFile Read)]

```

```
instance Handler FileIO IO
```

## 8.3 RND

```
module Effect.Random

import Effects
import Data.Vect
import Data.Fin

RND : EFFECT

srand  : Integer -> Eff m () [RND]
rndInt : Integer -> Integer -> Eff m Integer [RND]
rndFin : (k : Nat) -> Eff m (Fin (S k)) [RND]

instance Handler Random m
```

## 8.4 SELECT

```
module Effect.Select

import Effects

SELECT : EFFECT

select : List a -> Eff m a [SELECT]

instance Handler Selection Maybe
instance Handler Selection List
```

## 8.5 STATE

```
module Effect.State

import Effects

STATE : Type -> EFFECT

get    : Eff m x [STATE x]
put    : x -> Eff m () [STATE x]
putM   : y -> Eff m () [STATE x] [STATE y]
update : (x -> x) -> Eff m () [STATE x]

instance Handler State m
```

## 8.6 STDIO

```
module Effect.StdIO

import Effects
import Control.IOExcept
```

```
STDIO : EFFECT

putChar  : Handler StdIO m => Char -> Eff m () [STDIO]
putStr   : Handler StdIO m => String -> Eff m () [STDIO]
putStrLn : Handler StdIO m => String -> Eff m () [STDIO]

getStr   : Handler StdIO m => Eff m String [STDIO]
getChar  : Handler StdIO m => Eff m Char [STDIO]

instance Handler StdIO IO
instance Handler StdIO (IOExcept a)
```

## 8.7 SYSTEM

```
module Effect.System

import Effects
import System
import Control.IOExcept

SYSTEM : EFFECT

getArgs : Handler System e => Eff e (List String) [SYSTEM]
time    : Handler System e => Eff e Int [SYSTEM]
getEnv  : Handler System e => String -> Eff e (Maybe String) [SYSTEM]

instance Handler System IO
instance Handler System (IOExcept a)
```